# Automated Composition of Nondeterministic Stateful Services

Giuseppe De Giacomo and Fabio Patrizi

Dipartimento di Informatica e Sistemistica
SAPIENZA - Università di Roma
Via Ariosto 25 - 00185 Roma, Italy
{degiacomo,patrizi}@dis.uniroma1.it

**Abstract.** This paper addresses the automated composition of nondeterministic available services modeled as transition systems. Nondeterminism stems naturally when the results of client-service interactions cannot be foreseen, and calls for specific orchestration strategies able to deal with partial controllability. We show how to build a set of orchestrators, by resorting to a variant of the simulation relation's formal notion, by exploiting recent results on LTL formulas' synthesis and by reducing our technique to the search for a safety game winning strategy. The resulting technique is sound, complete and optimal w.r.t. computational complexity, and generates all possible solutions at once.

## 1 Introduction

Web services are modular applications that can be described, published, located, invoked and composed over a variety of networks (including the Internet): any piece of code and any application component deployed on a system can be wrapped and transformed into a network-available service, by using standard (XML-based) languages and protocols (e.g., WSDL, SOAP, etc.)- see e.g., [1]. The promise of Web services is to enable the composition of new distributed applications/solutions: when available services cannot satisfy a desired specification, they, or their parts, can be composed and orchestrated in order to realize the specification. Service composition involves two different phases [13]: the *composition synthesis*, where the specification of an orchestrator, which coordinates the available services to fulfill a target service specification, is synthesized, and the *composition deployment*, i.e., the actual implementation of the orchestrator specification in a given technology (such as BPEL) [1]. Here, we focus on the former.

Most of the research on composition synthesis, e.g., [27, 8], has considered *atomic* services, essentially abstracting away from their dynamic behavior (a.k.a. *possible conversations*). Notable exceptions are, e.g., [17, 7, 21, 5, 14, 20, 10] where stateful services, and their dynamic behavior, are considered explicitly. A survey on composition synthesis approaches can be found in [13].

[1] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

In this paper, we follow the general approach first proposed in [5], called, in [13], "Roman Model", and recently further investigated in the context of agent behavior composition, in AI [9, 25, 24]. We address the automatic composition of nondeterministic, partially controllable, available services, modeled as transition systems that capture the possible conversations that services can have with clients. When the result of interactions cannot be foreseen, nondeterminism naturally stems. For instance, think of a service for buying tickets: the service cannot know in advance whether seats are available for a selected performance. In other words, service behavior is partially controllable: a property an orchestrator needs to cope with. We assume that orchestrators can observe available services' states, and hence take advantage of this in choosing how to continue a certain task. This assumption is quite natural in this context [2], as transition systems represent available services' "public" behavior.

Our composition technique is based on the formal notion of simulation relation [18]. It follows the lines drawn in [6, 24], in the presence of nondeterminism, which calls for a specific simulation relation's variant, that considers available services' partial controllability. The variant presented here can be proven equivalent to the one in [24].

Our main contribution is relating the service composition problem to the literature on synthesis of reactive systems (cf., e.g., [22]). In particular, we show that the problem can be solved by exploiting safety-games and propose an implementation based on the system TLV. This is a major step toward practical implementation of engines for orchestrator synthesis. Notably, the proposed technique not only is sound, complete, and optimal w.r.t. computational complexity, but also, in a precise formal sense (see later), produces all (infinite) possible solutions at once.

In Section 2, we introduce the formal setting; in Section 3, we develop our technique; in Section 4, we show how the technique can exploit safety games; in Section 5, we propose an implementation based on the system TLV and, finally, in Section 6 we draw some conclusions.

## 2 The framework

The framework adopted here is based on [5, 9, 24], and is sometime referred to as the "Roman Model" [13].

**Data box**. We assume to have an accessible shared system, called *data box*, which allows client services to store and retrieve shared data. We describe it as a nondeterministic transition system, where *(i)* states represent an abstract finite description of the data content, and *(ii)* transitions represent the execution of operation, including data insertion/deletion or retrieval. Nondeterministic transitions model those operations whose outcome is a-priori unknown.

A *data box* is a tuple $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$ where:

[2] The reader should observe that also the standard proposal WSDL 2.0 has a similar point of view: the same operation can have multiple output messages (the `out message` and various `outfault messages`), and the client observes how the service behaved only after receiving a specific output message.

- $\mathcal{O}$ is the finite set of shared operations, i.e., the *whole* set of operations clients can perform, each of which may or may not affect data box' state;
- $D$ is the finite set of data box' states;
- $d_0 \in D$ is the initial state;
- $\rho \subseteq D \times \mathcal{O} \times D$ is the transition relation among states: $\langle d, o, d' \rangle \in \rho$, or $d \xrightarrow{o} d'$ in $\mathcal{DB}$, denotes that execution of operation $o$ in state $d$ may lead the data box to a successor state $d'$.

**Available Services**. An available service, at each step, offers to its clients a choice of operations, based upon its own and data box' state; the client chooses one of them, and the service executes it, resulting in a new state of the service and a new state of the data box. The available service can take into account data box' influence on available services by putting *guards* on transitions –i.e., conditions on current state of the databox, which restricts the set of transitions that can actually take place.

Formally, an *available service* over a data box $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$ is a tuple $\mathcal{S} = \langle \mathcal{O}, S, s_0, S^f, G, \varrho \rangle$, where:

- $\mathcal{O}$ is the same set of operations as in $\mathcal{O}$;
- $S$ is the finite set of service's states;
- $s_0 \in S$ is the initial state;
- $S^f \subseteq S$ is the set of *final* states, i.e., those where the execution can be legally stopped (if desired);
- $G$ is a set of boolean functions $g : D \to \{true, false\}$ called *guards*;
- $\varrho \subseteq S \times G \times \mathcal{O} \times S$ is the service's transition relation.

When $\langle s, g, o, s' \rangle \in \varrho$, we say that *transition* $s \xrightarrow{g,o} s'$ *is in* $\mathcal{S}$. Given a state $s \in S$, if there exists a transition $s \xrightarrow{g,o} s'$ in $\mathcal{S}$ (for some $g$ and $s'$) and the data box is in a state $d$ such that $g(d) = true$ then operation $o$ is said to be *executable* in $s$. A transition $s \xrightarrow{g,o} s'$ in $\mathcal{S}$ denotes that $s'$ is a possible successor state of $s$, when operation $o$ is executed in $s$, provided $g(d) = true$, $d$ being current data box state.

Available services are, in general, *nondeterministic*, that is, they allow many transitions to take place under execution of a same operation. So, when choosing the operation to execute next, the client of the service cannot be certain of which choices will be available later on, this depending on which transition actually takes place. In other words, nondeterministic behaviors are only *partially controllable*.

We say that a service $\mathcal{S}$ over a data box $\mathcal{DB}$ is *deterministic* iff there is no $\mathcal{DB}$ state $d \in D$ for which there exist, in $\mathcal{S}$, two distinct transitions $s \xrightarrow{g_1,o} s'$ and $s \xrightarrow{g_2,o} s''$ such that $s' \neq s''$ and $g_1(d) = g_2(d) = true$. Notice that given a deterministic service's state and a legal operation in that state, the *unique* next service state is always known. That is, deterministic services are indeed *fully controllable* by selecting operations.

**Community and Target Service**. A *community* $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ is a set containing *(i)* a data box $\mathcal{DB}$ and *(ii)* $n$ nondeterministic available services over $\mathcal{DB}$. In our framework, we also define the so-called *target service*, which is the *deterministic* service one aims at building by properly *composing* available services. A target service has the same form as any other service defined over $\mathcal{DB}$, with the only requirement of being deterministic.

**Trace and History**. Given $\mathcal{S} = \langle \mathcal{O}, S, s_0, S^f, G, \varrho \rangle$ over data box $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$, a *trace* for $\mathcal{S}$ on $\mathcal{DB}$ is a possibly infinite sequence, alternating configurations and operations, of the form $\langle s^0, d^0 \rangle \xrightarrow{o^1} \langle s^1, d^1 \rangle \xrightarrow{o^2} \cdots$, such that *(i)* $\langle s^0, d^0 \rangle = \langle s_0, d_0 \rangle$, and *(ii)* for all $j > 0$, if $\langle s^j, d^j \rangle \xrightarrow{o^{j+1}} \langle s^{j+1}, d^{j+1} \rangle$ , then $s^j \xrightarrow{g, o^{j+1}} s^{j+1}$ in $\mathcal{S}$ with $g(d^j) = true$ for some $g$, and $d^j \xrightarrow{o^{j+1}} d^{j+1}$ in $\mathcal{DB}$.

Similarly, let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ be a community, where $\mathcal{S}_i = \langle \mathcal{O}, S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ $(i = 1, \ldots, n)$ and $\mathcal{DB}$ as above. A *community trace* for $\mathcal{C}$ is a possibly infinite sequence of the form $\langle s_1^0, \ldots, s_n^0, d^0 \rangle \xrightarrow{o^1, k^1} \langle s_1^1, \ldots, s_n^1, d^1 \rangle \xrightarrow{o^2, k^2} \cdots$, such that *(i)* $\langle s_1^0, \ldots, s_n^0, d^0 \rangle = \langle s_{10}, \ldots, s_{n0}, d_0 \rangle$, and *(ii)* for all $j > 0$, if $\langle s_1^j, \ldots, s_n^j, d^j \rangle \xrightarrow{o^{j+1}, k^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1}, d^{j+1} \rangle$, then $s_{k^{j+1}}^j \xrightarrow{g, o^{j+1}} s_{k^{j+1}}^{j+1}$ in $\mathcal{S}_{k^{j+1}}$ with $g(d^j) = true$ for some $g$, $s_i^{j+1} = s_i^j$ for $i \neq k^{j+1}$, and $d^j \xrightarrow{o^{j+1}} d^{j+1}$ in $\mathcal{DB}$.

We call (community) *history* every finite prefix of a (community) trace ending with a configuration. Given a history $h$, we denote by $last(h)$ the last configuration, and by $length(h)$ the number of alternations between configurations and operations in $h$. Notice that the history of length 0 is simply the initial configuration of a trace (which is the same for every trace).

**Orchestrator**. The *orchestrator* is a component able to activate, stop and resume each of the available services, and select one to perform an executable operation. The orchestrator has *full observability* on available service states, that is, it can keep track (at runtime) of the current state of each available service. Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ be a community and $\mathcal{H}$ be the set of its community service histories. An *orchestrator* for a community $\mathcal{C}$ is a function $P : \mathcal{H} \times \mathcal{O} \to \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ and an operation $o \in \mathcal{O}$, selects an available service, i.e., returns its index, to which delegate $o$. Special value $u$ is introduced for technical convenience, to make function $P$ total.

**Definition 1.** *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$, where, $\mathcal{S}_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ $(i = t, 1 \ldots, n)$ and $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$. Let $P : \mathcal{H} \times \mathcal{O} \to \{1, \ldots, n, u\}$ be an orchestrator for $\mathcal{C}$. Given a trace $\tau = \langle s^0, d^0 \rangle \xrightarrow{o^1} \langle s^1, d^1 \rangle \xrightarrow{o^2} \cdots$ of $\mathcal{S}_t$ on $\mathcal{DB}$, we say that the orchestrator $P$ realizes the trace $\tau$ if and only if:*

- *for all community service histories $h \in \mathcal{H}_\tau$, $P(h, o^{length(h)+1}) \neq u$ and $\mathcal{H}_\tau^\ell \neq \emptyset$ (see below), where $\mathcal{H}_\tau = \bigcup_\ell \mathcal{H}_\tau^\ell$ is a set of community service histories, inductively defined as follows:*
  - $\mathcal{H}_\tau^0 = \{\langle s_{10}, \ldots, s_{n0}, d_0 \rangle\}$;
  - $\mathcal{H}_\tau^{j+1}$ *is the set of community histories of length $j+1$ having the form $h' = h \xrightarrow{o^{j+1}, k^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1}, d^{j+1} \rangle$ such that:*
    - *$h \in \mathcal{H}_\tau^j$, with $last(h) = \langle s_1^j, \ldots, s_n^j, d^j \rangle$;*
    - *$o^{j+1}$ and $d^{j+1}$ are the operation and the data box state in history of length $j+1$ obtained from $\tau$.*

- $P(h, o^{j+1})=k$, *that is, the orchestrator states that operation* $o^{j+1}$ *in the trace* $\tau$ *after community history* $h$ *should be executed by available service* $\mathcal{S}_k$*;*
- $s_k^j \xrightarrow{g,o^{j+1}} s_k^{j+1}$ *in* $\mathcal{S}_k$ *with* $g(d^j) = true$ *for some* $g$*, that is, the available service* $\mathcal{S}_k$ *can evolve according to the history* $h'$*.*
- $s_i^{j+1} = s_i^j$ *for each* $i \neq k$

– *if a configuration* $\langle s_t^\ell, d^\ell \rangle$ *of* $\tau$ *is such that* $s_t^\ell \in \mathcal{S}_t^f$*, then every configuration* $\langle s_1^\ell, \ldots, s_n^\ell, d^\ell \rangle = last(h)$*, with* $h \in \mathcal{H}_\tau^\ell$*, is such that* $s_i^\ell \in \mathcal{S}_i^f$*, for* $i = 1, \ldots, n$*.*

**Definition 2.** *An orchestrator* $P$ *for* $\mathcal{C}$ *is a composition of the target service* $\mathcal{S}_t$ *on data box* $\mathcal{DB}$ *iff it* realizes all traces *of* $\mathcal{S}_t$ *on* $\mathcal{DB}$*.*

Intuitively, the orchestrator realizes a target service if for all target service traces over the data box, at every step, it returns the index of an available service that can actually perform the requested operation. Observe that since available services and data box are nondeterministic, the orchestrator must be always able to execute the next operation, no matter how the activated service and the data box happen to evolve after each step. Finally, note that the orchestrator can observe available services' and data box' states (in fact, the whole community service history so far), in order to decide which available service to select next. This makes orchestrators akin to an advanced form of conditional plans [11].

**The Composition Problem**. This work addresses the following problem: *given a community* $\mathcal{C} = \{\mathcal{S}_1, \ldots, S_n, \mathcal{DB}\}$ *and a deterministic* target *service* $\mathcal{S}_t$ *over* $\mathcal{DB}$*, synthesize an orchestrator for* $\mathcal{C}$ *which is a composition of* $\mathcal{S}_t$ *on data box* $\mathcal{DB}$*.*

## 3 Composition via Simulation

Following [6, 24], we present a composition technique based on the formal notion of *simulation* [18, 12]. Since the devilish nondeterminism of both data box and available services prevents the possibility to use the off-the shelf notion of simulation, a more general variant is needed, called *ND-simulation*.

**Definition 3.** *Let* $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ *be a community and* $\mathcal{S}_t$ *a target service over* $\mathcal{DB}$*, where,* $\mathcal{S}_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ *(*$i = t, 1 \ldots, n$*) and* $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$*. An ND-simulation relation of* $\mathcal{S}_t$ *by* $\mathcal{C}$ *is a relation* $R \subseteq S_t \times S_1 \times \ldots \times S_n \times D$ *such that* $\langle s_t, s_1, \ldots, s_n, d \rangle \in R$ *implies:*

1. *if* $s_t \in S_t^f$ *then* $s_i \in S_i^f$*, for* $i = 1, \ldots, n$*;*
2. *for each* $o \in \mathcal{O}$*, there exists a* $k \in \{1, \ldots, n\}$ *such that for all* $\langle s_t, d \rangle \xrightarrow{o} \langle s_t', d' \rangle$ *such that* $s_t \xrightarrow{g,o} s_t'$ *in* $\mathcal{S}_t$ *with* $g(d) = true$ *and* $d \xrightarrow{o} d'$ *in* $\mathcal{DB}$ *, then both the followings hold:*
   (a) *there exists a transition* $s_k \xrightarrow{g,o} s_k'$ *in* $\mathcal{S}_k$ *with* $g(d) = true$*;*
   (b) *for all* $s_k \xrightarrow{g,o} s_k'$ *in* $S_k$ *with* $g(d) = true$ *we have that* $\langle s_t', s_1, \ldots, s_k', \ldots, s_n, d' \rangle \in R$*.*

An ND-simulation is essentially a simulation between $\mathcal{S}_t$ and the asynchronous product of the services $\mathcal{S}_i$ in $\mathcal{C}$. With respect to the usual notion of simulation relation, we need to deal with data box $\mathcal{DB}$ in $\mathcal{C}$ that acts as a parameter, and, more importantly, we need to take into account available services' nondeterminism. To this end, we require that *(i)* for each target service's transition an available service $k$ can be selected to perform $\mathcal{S}_t$ labeling operation and *(ii) all possible successor states* (under selected service and current operation) are still included in the ND-simulation relation.

A state $s_t$ *is ND-simulated by* $\langle s_1, \ldots, s_n, d\rangle$ (or $\langle s_1, \ldots, s_n, d\rangle$ *ND-simulates* $s_t$), denoted $s_t \preceq \langle s_1, \ldots, s_n, d\rangle$, iff there exists an ND-simulation $R$ of $\mathcal{S}_t$ by $\mathcal{C}$ such that $\langle s_t, s_1, \ldots, s_n, d\rangle \in R$. Observe that this is a coinductive definition. As a result, the relation $\preceq$ is itself an ND-simulation, and is in fact the *largest ND-simulation relation*.

Next result shows that checking for the existence of a composition can be reduced to checking whether there exists an ND-simulation relation between the target service and the community, containing their respective initial states.

**Theorem 1.** *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$ as above. An orchestrator $P$ for $\mathcal{C}$ that is a composition of target service $S_t$ over $\mathcal{DB}$ exists if and only if $s_{t0} \preceq \langle s_{10}, \ldots, s_{n0}, d_0\rangle$.*

Theorem 1 provides a straightforward method to check for the existence of a composition, namely:

1. compute the largest ND-simulation relation $\preceq$;
2. check whether $\langle s_{t0}, s_{10}, \ldots, s_{n0}, d_0\rangle \in \preceq$.

From the computational point of view, the largest ND-simulation relation $\preceq$ between $\mathcal{S}_t$ and $\mathcal{C}$ can be computed in polynomial time wrt the size of $\mathcal{S}_t$ and $\mathcal{C}$. Since the number of states in $\mathcal{C}$ is exponential in the number of available services $n$, $\preceq$ can be computed in exponential time. More precisely, it is polynomial wrt the size of $\mathcal{S}_t$, $\mathcal{DB}$ and each service $\mathcal{S}_i$, but exponential in the number of available services $n$. Thus, observing that the problem is EXPTIME-hard [19], we get that this technique is optimal wrt worst-case complexity.

Once we have computed the ND-simulation, *synthesizing* an orchestrator becomes an easy task. As a matter of fact, there is a well-defined procedure that, given an ND-simulation, builds a finite state program that returns, at each point, the set of available behaviors capable of performing a target-conformant operation. We call such a program *orchestrator generator*, or simply *PG*. Formally:

**Definition 4.** *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$ as above. The orchestrator generator (PG) of $\mathcal{C}$ for $\mathcal{S}_t$ is a tuple $PG = \langle \mathcal{O}, \{1, \ldots, n\}, \Sigma, \partial, \omega\rangle$, where:*

1. *$\mathcal{O}$ is the finite set of operations;*
2. *$\{1, \ldots, n\}$ is the set of available behavior indexes;*
3. *$\Sigma = \{\langle s_t, s_1, \ldots, s_n, d\rangle \mid s_t \preceq \langle s_1, \ldots, s_n, d\rangle\}$ is the set of states of PG, formed by the tuples belonging to the largest ND-simulation relation;*
4. *$\partial \subseteq \Sigma \times \mathcal{O} \times \{1, \ldots, n\} \times \Sigma$ is the* transition relation, *where $\langle \sigma, o, k, \sigma'\rangle \in \partial$, or $\sigma \xrightarrow{o,k} \sigma'$* is in *PG, if and only if all of the followings hold:*
   - *$\sigma = \langle s_t, s_1, \ldots, s_k, \ldots, s_n, d\rangle$ and $\sigma' = \langle s'_t, s_1, \ldots, s'_k, \ldots, s_n, d'\rangle$*
   - *$s_t \xrightarrow{g,o} s'_t$ in $\mathcal{S}_t$ with $g(d) = true$;*

- *there exists a transition* $s_k \xrightarrow{g,o} s'_k$ *in* $\mathcal{S}_k$ *with* $g(d) = true$*;*
- *for all transitions* $s_k \xrightarrow{g,o} s''_k$ *in* $\mathcal{S}_k$ *with* $g(d) = true$ *we have* $\langle s'_t, s_1, \ldots, s''_k, \ldots, s_n, d' \rangle \in \Sigma$*;*

5. $\omega : \Sigma \times \mathcal{O} \mapsto 2^{\{1,\ldots,n\}}$ *is the* output function, *where:*
   - $\omega(\sigma, o) = \{k \mid \exists\ \sigma'$ *s.t.* $\sigma \xrightarrow{o,k} \sigma'$ *in PG*$\}$.

Intuitively, *PG* is a finite state transducer that, given an operation $o$ (compliant with the target service), outputs, through $\omega$, the set of *all* available services able to perform $o$ next, according to the largest ND-simulation $\preceq$. Observe that computing *PG* from the relation $\preceq$ is easy, since it involves checking for *local* conditions only.

If there exists a composition of $\mathcal{S}_t$ by $\mathcal{C}$, then $s_{t0} \preceq \langle s_{10}, \ldots, s_{n0}, d_0 \rangle$ and *PG* does include state $\sigma_0 = \langle s_{t0}, s_{10}, \ldots, s_{n0}, d_0 \rangle$. In such case, all the actual orchestrators that are compositions of $\mathcal{S}_t$ by $\mathcal{C}$ can be obtained by just picking up, at each step, one among the services returned by $\omega$. Being, in fact, generated from a given structure (i.e., *PG*), they are called *generated orchestrators*. Prior to provide their formal definition, some preliminary notions are needed.

A *trace for PG* starting from $\sigma^0$ is a finite or infinite sequence of the form $\sigma^0 \xrightarrow{o^1,k^1} \sigma^1 \xrightarrow{o^2,k^2} \cdots$, such that $\sigma_j \xrightarrow{o^{j+1},k^{j+1}} \sigma_{j+1}$ is in *PG*, for all $j$. A *history for PG* starting from state $\sigma^0$ is a prefix of a trace starting from state $\sigma^0$. By using histories, one can introduce *PG-orchestrators*, which are functions $PGP_{\text{CHOOSE}} : \mathcal{H}_{PG} \times \mathcal{O} \to \{1, \ldots, n, u\}$ where $\mathcal{H}_{PG}$ is the set of *PG* histories starting from any state in $\Sigma$ and defined as follows: $PGP_{\text{CHOOSE}}(h_{PG}, o) = \text{CHOOSE}(\omega(last(h_{PG}), o))$, for all $h_{PG} \in \mathcal{H}_{PG}$, where CHOOSE stands for a choice function that chooses one element among those returned by $\omega(last(h_{PG}), o))$.

We can now relate a *PG* to compositions, through the following characterizing theorem.

**Theorem 2.** *If PG includes the state* $\sigma_0 = \langle s_{10}, \ldots, s_{n0}, d_0 \rangle$ *then every orchestrator generated by PG is a composition of the target service* $\mathcal{S}_t$ *by the community* $\mathcal{C}$*. Moreover, every orchestrator that is a composition of the target service* $\mathcal{S}_t$ *by the community* $\mathcal{C}$ *can be generated by PG (which, indeed, includes* $\sigma_0$*).*

Notably, while each specific composition may be an infinite state program, *PG*, which includes them all, is always finite. We conclude the section with an interesting observation. Let us consider the generated orchestrator $PGP_{jit}$, with CHOOSE resolved at run-time. $PGP_{jit}$ (and *PG* for the matter) can be computed *on-the-fly* by storing only the ND-simulation $\preceq$. Indeed, at each point, the only information we need for the next choice is $\omega(\sigma, o)$ where $\sigma \in \Sigma = \preceq$. Now, in order to compute $\omega(\sigma, o)$ we only need to know $\preceq$.

## 4 Simulation and Safety Games

In this Section, we show how a service composition problem instance can be encoded into a game structure and how searching for a composition is equivalent to searching for a winning strategy for the corresponding game (cf. [3, 4, 22]). The main motivation

behind this approach is the increasing availability of software systems, such as TLV [23], Lily [15], Anzu [16] or MOCHA [2], which provide *(i)* efficient procedures for strategy computation and *(ii)* convenient languages for representing the problem instance in a modular, intuitive and straightforward way.

### 4.1 Safety-Game structures

We specialize the *game structures* proposed in [22] to deal with synthesis problems for invariant properties. Throughout the rest of the paper, we assume to deal with infinite-run TSs, possibly obtained by introducing fake loops, as customary in LTL verification/synthesis.

Starting from [22], we define a *safety-game structure* (or $\Box$-game structure or $\Box$-GS, for short) as a tuple $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$, where:

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of *state* variables, ranging over *finite* domains $V_1, \ldots, V_n$, respectively. $V = V_1, \ldots, V_n$ represents the set of all possible valuations of variables in $\mathcal{V}$. We assume that $\mathcal{V} = \{\mathcal{X}, \mathcal{Y}\}$, i.e., $\mathcal{V}$ is partitioned into sets $\mathcal{X}$ and $\mathcal{Y}$, the former referred to *as set of environment variables* and the latter as *set of system variables*. Let $X$ (resp. $Y$) be the set of all possible valuations for variables in $\mathcal{X}$ ($\mathcal{Y}$). Then, $\boldsymbol{x} \in X$ ($\boldsymbol{y} \in Y$) is called *environment state* (*system state*). A *game state* $\boldsymbol{s} \in V$ is a complete assignment of values to variables. Without loss of generality, we assume that $\boldsymbol{s} = \langle \boldsymbol{x}, \boldsymbol{y} \rangle \in X \times Y$.
- $\Theta$ is a formula representing the initial states of the game. It is a boolean combination of expressions $(v_k = \bar{v}_k)$, where $v_k \in \mathcal{V}$ and $\bar{v}_k \in V_k$ ($k \in \{1, \ldots, n\}$) (partial assignments are allowed). For such formulae, given a state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V$, we write $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \models \Theta$ if state $s$ satisfies the assignments specified by $\Theta$.
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the *environment transition relation* which relates a current (unprimed) game state to a possible next (primed) environment state.
- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the *system transition relation*, which relates a game state plus a next environment state to a next system state.
- $\Box\varphi$ is a formula representing the invariant property to be guaranteed, where $\varphi$ has the same form as $\Theta$.

We assume variables in $\mathcal{X}$ (respectively $\mathcal{Y}$) are ordered, so that valuations in $X$ ($Y$) can be conveniently represented as tuples $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ ($\boldsymbol{y} = \langle y_1, \ldots, y_m \rangle$). In unary tuples, we omit angle brackets when no ambiguity arises.

A game state $\langle \boldsymbol{x}', \boldsymbol{y}' \rangle$ is a *successor* of $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ iff $\rho_e(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}')$ and $\rho_s(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}')$. A *play* of $G$ is a maximal sequence of states $\eta : \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \langle \boldsymbol{x}_1, \boldsymbol{y}_1 \rangle \cdots$ satisfying *(i)* $\langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \models \Theta$, and *(ii)* for each $j \geq 0$, $\langle \boldsymbol{x}_{j+1}, \boldsymbol{y}_{j+1} \rangle$ is a successor of $\langle \boldsymbol{x}_j, \boldsymbol{y}_j \rangle$. Given a $\Box$-GS $G$, in a given state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ of a game play, the environment chooses an assignment $\boldsymbol{x}' \in X$ such that $\rho_e(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}')$ holds and the system chooses assignment $\boldsymbol{y}' \in Y$ such that $\rho_s(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}')$ holds.

A play is said to be *winning for the system* if it is infinite and satisfies the winning condition $\Box\varphi$. Otherwise, it is *winning for the environment*. A *strategy* for the system is a partial function $f : (X \times Y)^+ \times X \to Y$ such that for every $\lambda : \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \cdots \langle \boldsymbol{x}_n, \boldsymbol{y}_n \rangle$ and for every $\boldsymbol{x}' \in X$ such that $\rho_e(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{x}')$, $\rho_s(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{x}', f(\lambda, \boldsymbol{x}'))$ holds. A play

$\eta : \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \langle \boldsymbol{x}_1, \boldsymbol{y}_1 \rangle \cdots$ is said to be *compliant* with a strategy $f$ iff for all $i \geq 0$, $f(\langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \cdots \langle \boldsymbol{x}_i, \boldsymbol{y}_i \rangle, \boldsymbol{x}_{i+1}) = \boldsymbol{y}_{i+1}$. A strategy $f$ is winning for the system from a given state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ iff all plays starting from $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ and compliant with $f$ are so. When such a strategy exists, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ is said to be a *winning state* for the system. A $\Box$-GS is said to be *winning for the system* if all initial states are so. Otherwise, it is said to be *winning for the environment*.

Our objective is to encode a composition problem into a $\Box$-GS and, then, exploit tools available for the latter to compute the orchestrator generator $PG$ (cf. Section 3). Essentially, as it will be clear soon, one can extract the maximal ND-simulation relation –and, from this, directly compute the $PG$–, from the maximal set of states that are *winning* for the system. Let us show how such *winning set* can be computed in general on a $\Box$-GS. The core of the algorithm is the following operator (cf. [3, 22]):

**Definition 5.** *Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ be a $\Box$-GS as above. Given a set $P \subseteq V$ of game states $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$, the set of $P$*'s controllable predecessors *is*

$$\pi(P) \doteq \{\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V \mid \forall\, \boldsymbol{x}'.\rho_e(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}') \rightarrow \exists\, \boldsymbol{y}'.\rho_s(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}') \wedge \langle \boldsymbol{x}', \boldsymbol{y}' \rangle \in P\}$$

Intuitively, $\pi(P)$ is the set of states from which the system can force the play to reach a state in $P$, no matter how the environment evolves. Based on this, Algorithm 1 computes the set of all system's winning states of a $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$, as Theorem 3 shows.

**Algorithm 1** $WIN$ – Computes system's maximal set of winning states in a $\Box$-GS

1: $W := \{\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V \mid \langle \boldsymbol{x}, \boldsymbol{y} \rangle \models \varphi\}$
2: **repeat**
3: $\quad W' := W$;
4: $\quad W := W \cap \pi(W)$;
5: **until** $(W' = W)$
6: **return** $W$

**Theorem 3.** *Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ be a $\Box$-GS as above and $W$ be obtained as in Algorithm 1. Given a state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V$, a system's winning strategy $f$ starting from $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ exists iff $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in W$.*

In fact, one can define a system's winning strategy $f(\langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle, \ldots, \langle \boldsymbol{x}_i, \boldsymbol{y}_i \rangle, \boldsymbol{x}) = \boldsymbol{y}$, by picking up, for each $\boldsymbol{x}$ such that $\rho_e(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{x})$ holds, any $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in W$.

### 4.2 From Composition to Safety Games

In order to encode the composition problem as a $\Box$-GS, we need first to individuate which place each abstract component, e.g., target, available services, data box, occupies in the game representation. Conceptually, our goal is to refine an automaton capable of selecting, at each step, one among all the available services, in a way such that

the community is always able to satisfy target service requests. So, the orchestrator, i.e., the object of the synthesis, plays as system and, consequently, the other entities, properly combined, form the environment. In addition, according to our purposes, the winning condition requires to satisfy two properties: *(i)* if the target service is in a final state, all community services are in a final state as well; *(ii)* the service selected by the orchestrator is able to perform the action currently requested by the target service.

Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$, where, $\mathcal{S}_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ ($i = t, 1 \ldots, n$) and $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$. We derive a $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_c, \Box\varphi \rangle$, as follows:

- $\mathcal{V} = \{s_t, s_1, \ldots, s_n, d, o, ind\}$, where:
  - $s_i$ ranges over $S_i \cup \{init\}$ ($i = t, 1, \ldots, n$);
  - $d$ ranges over $D \cup \{init\}$;
  - $o$ ranges over $\mathcal{O} \cup \{init\}$;
  - $ind$ ranges over $\{1, \ldots, n\} \cup \{init\}$;

  with an intuitive semantics: each complete valuation of $\mathcal{V}$ represents *(i)* the current state of community (variables $s_1, \ldots, s_n$,), data box ($d$) and target service ($s_t$), *(ii)* the operation to be performed next ($o$) and *(iii)* the available service selected to perform it ($ind$). Special value $init$ has been introduced for convenience, so as to have fixed initial state;
- $\mathcal{X} = \{s_t, s_1, \ldots, s_n, d, o\}$ is the set of environment variables;
- $\mathcal{Y} = \{ind\}$ is the (singleton) set of system variables;
- $\Theta = (\bigwedge_{i=t,0,\ldots,n}(s_i = init)) \wedge (d = init) \wedge (o = init) \wedge (ind = init)$;
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is defined as follows:
  - $\langle \langle init, \ldots, init \rangle, init, \langle s_t, s_1, \ldots, s_n, d, o \rangle \rangle \in \rho_e$ iff $s_i = s_{i0}$, for $i = t, 1, \ldots, n$, $d = d_0$, and there exists a transition $\langle s_{t0}, g, o, s_t' \rangle \in \varrho_t$ such that $g(d_0) = true$;
  - if $s_i \neq init$, with $i = t, 1, \ldots, n,$, $d \neq init$, $o \neq init$ and $ind \neq init$ then $\langle \langle s_t, s_1, \ldots, s_n, d, o \rangle, ind, \langle s_t', s_1', \ldots, s_n', d', o' \rangle \rangle \in \rho_e$ iff the followings hold in conjunction:
    1. there exists a transition $s_t \xrightarrow{g,o} s_t'$ in $\varrho_t$ with $g(d) = true$;
    2. either there exists a transition $s_{ind} \xrightarrow{g,o} s_{ind}'$ in $\varrho_{ind}$ with $g(d) = true$ or $s_{ind}' = s_{ind}$ (service wrongly makes no move, and the error violates the safety condition $\varphi$, see below);
    3. $s_i = s_i'$, for all $i = 1, \ldots, n$ such that $i \neq ind$;
    4. there exists a transition $d \xrightarrow{o} d'$ in $\mathcal{DB}$;
    5. there exists a transition $s_t' \xrightarrow{g',o'} s_t''$ in $\varrho_t$ for some $s_t''$, with $g'(d') = true$;
- $\langle \langle s_t, s_1, \ldots, s_n, d, o \rangle, ind, \langle s_t', s_1', \ldots, s_n', d', o' \rangle, ind' \rangle \in \rho_s$ iff $ind' \in \{1, \ldots, n\}$;
- Formula $\varphi$ is defined depending on current state, operation and service selection as

$$\varphi \doteq \Theta \vee (\bigwedge_{i=1}^{n} \neg fail_i) \wedge (final_t \rightarrow \bigwedge_{i=1}^{n} final_i),$$

  where:
  - $fail_i \doteq (ind = i) \wedge (\bigwedge_{\langle s,g,op,s' \rangle \in \varrho_i}(g(d) = false \vee s_i \neq s \vee op \neq o))$, encodes the fact that service $i$ has been selected but, in its current state, no transition can take place which executes the requested operation;

- $final_i \doteq \bigvee_{s \in S_i^f}(s_i = s)$ encodes the fact that service $i = t, 1, \ldots, n$ is currently in one of its final states.

We can now show how the so-obtained game structure allows for computing an orchestrator generator. Recall that, in order to define the $PG$, one needs to build an ND-simulation (see Definition 4). The following Theorem shows that this can be equivalently done by computing the maximal system's set of winning states for $G$.

**Theorem 4.** *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$ where $\mathcal{S}_i = \langle \mathcal{O}, S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ ($i = t, 1 \ldots, n$) and $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$. From $\mathcal{C}$ and $\mathcal{S}_t$ derive: a $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ as shown above. Let $W \subseteq V$ be the maximal set of system's winning states for $G$. Then $\langle init, \ldots, init \rangle \in W$ if and only if $s_{t0} \preceq \langle s_{10}, \ldots, s_{n0}, d_0 \rangle$.*

Based on this, the following Theorem gives us an actual procedure to build up an orchestrator generator and, hence, all possible compositions.

**Theorem 5.** *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$, $\mathcal{S}_t$ and $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_c, \Box\varphi \rangle$ be as above (hypothesis of Theorem 4). Let $W$ be the system's winning set for $G$ with $\langle init, \ldots, init \rangle \in W$. Then the orchestrator generator $PG = \langle \mathcal{O}, \{1, \ldots, n\}, \Sigma, \partial, \omega \rangle$ of $\mathcal{C}$ for $\mathcal{S}_t$ can be built from $W$, as follows:*

- *$\mathcal{O}$ is the usual set of operations and $\{1, \ldots, n\}$ the set of available services' indexes;*
- *$\Sigma \subseteq S_t \times S_1 \times \ldots \times S_n \times D$ is such that $\langle s_t, s_1, \ldots, s_n, d \rangle \in \Sigma$ if and only if there exists a game state $\langle s_t, s_1, \ldots, s_n, d, o, ind \rangle \in W$, for some $o \in \mathcal{O}$ and $ind \in \{1, \ldots, n\}$;*
- *$\partial \subseteq (\Sigma \times \mathcal{O} \times \{1, \ldots, n\} \times \Sigma)$ is such that $\langle \langle s_t, s_1, \ldots, s_n, d \rangle, o, k, \langle s_t', s_1', \ldots, s_n', d' \rangle \rangle \in \partial$ if and only if $\langle s_t, s_1, \ldots, s_n, d, o, k \rangle \in W$ and there exist $o' \in \mathcal{O}$ and $k' \in \{1, \ldots, n\}$ such that $\langle \langle s_t, s_1, \ldots, s_n, d, o, k \rangle, \langle s_1', \ldots, s_n', s_t', d', o', k' \rangle \rangle \in \rho_s$;*
- *$\omega : \Sigma \times \mathcal{O} \to 2^{\{1, \ldots, n\}}$ is defined as $\omega(\langle s_1, \ldots, s_n, s_t, d \rangle, o) = \{i \in \{1, \ldots, n\} \mid \langle s_1, \ldots, s_n, s_t, d, o, i \rangle \in W\}$.*

The above theorems show how one can exploit tools from system synthesis for computing all compositions of a given target service. In details, starting from $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ and $\mathcal{S}_t$ one can build the corresponding game structure $G$, then compute the set $W$ and, if it contains $G$'s initial state, use such set to generate the $PG$. In fact, this last step is not really needed. Indeed, it is not hard to convince oneself that given a current state $\langle s_t, s_1, \ldots, s_n, d \rangle$ and an operation to be executed $o \in \mathcal{O}$, a service selection $ind$ is "good" (i.e, the selected service can actually execute the operation and the whole community can still simulate the target service) if and only if $W$ contains a tuple $\langle s_t, s_1, \ldots, s_n, d, o, ind \rangle$, for some $ind \in \{1, \ldots, n\}$. Consequently, at each step, on the basis of the current state $s_t$ of the target service, the states $s_1, \ldots, s_n$ of available services, the state $d$ of data box, and the operation $o$ requested, one can select a tuple from $W$, extract the $ind$ component, and use it for next service selection.

Finally, observe that time complexity of Algorithm 1 in polynomial in $|V|$, that is the size of input $\Box$-GS' state space. Since in our encoding $|V|$ is polynomial in $|S_1|, \ldots, |S_n|, |S_t|, |D|$ and exponential in $n$, we get:

**Theorem 6.** *Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$. Checking the existence of compositions by reduction to safety games can be done in polynomial time wrt $|S_1|, \ldots, |S_n|, |S_t|, |D|$ and exponential time in $n$.*

That is, the technique is actually optimal wrt worst-case time complexity, the composition problem being EXPTIME-hard [19].

## 5 Using TLV for computing compositions

Searching for a winning strategy is a problem solvable by several implemented systems (e.g., [16, 15, 23]). We focus on TLV [23], the basic concepts being valid for all others.

TLV is a software for verification and synthesis of LTL specifications, based on symbolic manipulation of states, by using Binary Decision Diagrams (BDDs). It takes two inputs: *(i)* a synthesis procedure and *(ii)* an LTL specification, encoded in SMV [23], to be manipulated by the procedure. In particular, we refer to a TLV-BASIC procedure for safety games which takes as input an LTL specification that encodes a $\Box$-GS and derives from the system's maximal winning set, if non empty, a structure representing the PG, as shown in Theorem 5. For a detailed description of TLV, TLV-BASIC and SMV, we refer to [23], here introducing some essentials only.

Our approach consists in deriving, from the composition problem specification, i.e., community and target service, the SMV encoding of the respective $\Box$-GS, as shown in Section 4.2, then execute TLV against this input and obtain, if the problem is feasible, the respective PG.

Figure 1 shows the basic blocks of a sample encoding for a composition problem with 3 available services. Module `Main` wraps up all other modules and represents the whole game. It consists of two submodules (here declared as `system`), `sys` and `env`, which encode, respectively, the environment and the system in the game structure. Goal formula `good` (i.e., the invariant property) is a combination of subformulae `initial` and `failure` of modules `sys` and `env`, directly obtained from the goal formula in the $\Box$-GS representation. Observe that `env` and `sys` evolve synchronously, the former choosing the operation and the latter selecting the service for its execution. The transition relation in module `Sys` encodes an *unconstrained controller*, able to output, at each step, any available service index in the interval $[1, n]$. The synthesis' objective is to restrict such a relation so to obtain a winning strategy.

As for module `Env`, it contains all basic blocks the $\Box$-GS environment consists of. Observe that its behavior depends on the value of module `sys`' `index` variable, as prescribed by `Main`. According to SMV semantics, modules `db`, `target`, `s1`, $\ldots$, `s`$n$ execute synchronously. However, each of them can be encoded so to emulate asynchrony, by looping when not selected. In particular, the encoding is such that, at each step, `db`, `target` and only one among `s1`, `s2`, $\ldots$, `s`$n$ move, according to the $\Box$-GS description. `Env` behavior is as follows. At each step, the available service selected by the current value of `index`, executes the operation requested by `target`, which is stored in `operation`. All other services loop in their current state. At the same time, `db` moves according to `operation`, `target` selects next operation, according to its specification, and `sys` selects a new service. Note that, in general, there may exist states where the selected service cannot perform the requested operation, due to

either operation precondition failure (i.e., `db` state) or service's current state. In such cases, expression `failure` of selected service becomes $true$ and, consequently, so does `env.failure`. Avoiding such situations, by properly constraining `sys` transition relation, is exactly the synthesis procedure aim.

```
MODULE Main
VAR
  env: system Env(sys.index);
  sys: system Sys;
DEFINE
  good :=
     (sys.initial & env.initial)|
      !(env.failure);
```

```
MODULE Sys
VAR
  index : 0..3; --num of services, 0 used for init
INIT
  index = 0
TRANS
  case
    index=0 : next(index)!=0;
    index!=0 : next(index)!=0;
  esac
DEFINE
  initial := (index=0);
```

```
MODULE Env(index)
 VAR
  operation  : {start_op,pick,store,play,display_content,free_mem};
  db : Databox(operation);
  target : Target(operation,db.state);
  s1 : Service1(index,operation,db.state);
  s2 : Service2(index,operation,db.state);
  s3 : Service3(index,operation,db.state);
DEFINE
  initial := (db.initial & s1.initial & s2.initial & s3.initial &
                              target.initial & operation=start_op);
  failure := (s1.failure | s2.failure | s3.failure) |
                      (target.final & !(s1.final & s2.final & s3.final));
```

**Fig. 1.** A TLV sample fragment encoding

## 6 Conclusions

We presented a new technique for composition of partially controllable available services, which exploits the relationships between *(i)* building a simulation relation and *(ii)* checking invariant properties in temporal-logic-based model checkers and synthesis systems (cf., e.g., [26, 4]). We showed that all compositions can be computed at once, as solutions to safety games and developed an implementation for the synthesis system TLV (http://www.cs.nyu.edu/acsys/tlv/ and cf., e.g., [22]). Another option would be to exploit ATL-based verifiers, such as Mocha (http://www.cis.upenn.edu/∼mocha/), which can check game-structures for properties such as invariants, and extract winning strategies for them.

## References


1. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijai Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer, 2004.

2. Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Proc. of CAV 1998*, pages 521–525, 1998.
3. Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20. Springer-Verlag, 1995.
4. Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.
5. Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proc. of ICSOC 2003*, pages 43–58, 2003.
6. Daniela Berardi, Fahima Cheikh, Giuseppe De Giacomo, and Fabio Patrizi. Automatic service composition via simulation. *Int. J. Found. Comput. Sci.*, 19(2):429–451, 2008.
7. Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. of WWW 2003*, 2003.
8. J. Cardose and A.P. Sheth. Introduction to semantic web services and web process composition. In *Proc. of SWSWPC 2004*, 2004.
9. Giuseppe De Giacomo and Sebastian Sardiña. Automatic synthesis of new behaviors from a library of available behaviors. In *Proc. of IJCAI 2007*, pages 1866–1871, 2007.
10. C.E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proc. of ICSOC 2004*, 2004.
11. Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kauffman, 2004.
12. Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of FOCS 1995*, pages 453–462, 1995.
13. Richard Hull. Web services composition: A story of models, automata, and logics. In *Proc. of SCC 2005*, 2005.
14. Richard Hull, Michael Benedikt, Vassilis Christophides, and Jianwen Su. E-Services: a Look Behind the Curtain. In *Proc. of PODS 2003*, pages 1–14, 2003.
15. Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Proc. FMCAD '06*, pages 117–124, 2006.
16. Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *Proc. of CAV 2007*, pages 258–262, 2007.
17. Sheila McIlraith and Tran Cao Son. Adapting Golog for programming the semantic web. In *Proc. of KR-02*, 2002.
18. Robin Milner. An algebraic definition of simulation between programs. In *Proc. of IJCAI 1971*, pages 481–489, 1971.
19. Anca Muscholl and Igor Walukiewicz. A lower bound on web services composition. In *Proc. of FoSSaCS 2007*. Springer, 2007.
20. Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, 2007.
21. M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proc. of ICWS 2005*, 2005.
22. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.
23. A. Pnueli and E. Shahar. The TLV system and its applications. Technical report, Weizmann Institute, 1996.
24. Sebastian Sardiña, Giuseppe De Giacomo, and Fabio Patrizi. Behavior composition in the presence of failure. In *Proceedings of KR'08*, 2008.

25. Sebastian Sardiña, Fabio Patrizi, and Giuseppe De Giacomo. Automatic synthesis of a global behavior from multiple distributed behaviors. In *Proc. of AAAI 2007*, pages 1063–1069, 2007.
26. M. Vardi and K. Fisler. Bisimulation and model checking. In *Proc of. CHARME*, pages 338–341, 1999.
27. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. of ISWC 2003*, 2003.